Visualising interactive inferences with IDPD3

Ruben Lapauw, Ingmar Dasseville, Marc Denecker

KU Leuven

Abstract. A large part of the use of knowledge base systems is the interpretation of the output by the end-users and the interaction with these users. Even during the development process visualisations can be a great help to the developer. We created IDPD3 as a library to visualise models of logic theories. IDPD3 is a new version of ID^P_{Draw} and adds support for visualised interactive simulations.

1 Introduction

Current logic inference systems often communicate with the user in a text-based method. Interpreting output of the system, e.g. in the form of a structure of an answer set or a structure, is often difficult for the user. Appropriate visualisations of the output can be an enormous help to the user. However creating a fitting visualisation is a cumbersome task. Therefore in the past, several system were developed to build visualisations with logic inference engines where logic itself is used to describe the visualisation (e.g. Kara [5],$ASPVIZ$ [3] and ID^P_{Draw} [6]).

These systems generally take an answer set containing special graphical facts. The combination of these facts is interpreted by the system to produce the visualisation. Kara extends this approach with a generalized visualisation approach that can visualise any answer set as a hypergraph. Kara also allows some interaction with the user by using abductive reasoning. This abductive reasoning will modify the initial interpretation to match the visualised output. This interactivity however cannot be used as a simulation.

IDPD3 is the successor of ID^P_{Draw}. Unlike ID^P_{Draw} which was written in C++, IDPD3 is written in Lua and Javascript with supporting libraries in both languages: a JSON encoder for Lua and the d3 visualisation library [2]. IDPD3 is now fully integrated with IDP and the IDP Web-IDE. This change allows for a better portability and maintenance. It also allows easier development of applications.

IDPD3 keeps most of the original features like drawing multiple frames of rectangles and circles. It also keeps the possibility to visualise graphs with an automatically generated layout. However some functions like polygons are not yet implemented. The original system is extended with basic animations when transitioning between two frames. And more importantly IDPD3 has support for interactively visualised linear time calculus theories.

In Bogaerts, Jansen, Bruynooghe, De Cat, Vennekens, and Denecker [1] a system is described that takes as input a logic linear time calculus theory that

describes a domain of interactive processes. The system then simulates an interactive process satisfying this theory using in a loop of user input and the progression inference to decide the next state. So far this system was text-based. The IDPD3 library allows to build a graphical interface to this system, visualising the current state and capturing user interaction with this visualisation to determine the next state.

In the following sections we describe IDP as a knowledge base system, introduce IDPD3 with its features and implementation and show how to create a full interactive simulation of a linear time theory. After this we will compare IDPD3 to other visualisation systems and describe some future extensions.

2 IDP

Input to IDP typically consists of four types of objects: a vocabulary (Σ), a theory ($\mathcal{T}$), a structure (S) and procedures. The first three declare the knowledge. The last is an imperative method to run inferences on the declared knowledge. An example of each can be found in Listing 1.

A vocabulary object has the syntactical form:

```
vocabulary <name> { <list of symbols>}
```

This declares a named vocabulary consisting of a set of symbols: types, typed predicates and typed functions.

A theory object has the syntactical form:

```
theory <name> : <voc name> { <list of sentences> }
```

This form declares a named theory over a vocabulary with a given set of sentences. The symbols used in the sentences must be a part of the vocabulary. The language of sentences in IDP supports the classical logical operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Rightarrow$, $\Leftrightarrow$). IDP extends this with predicates and quantifications ($\forall x / \exists x : \phi(x)$), with functions, arithmetic and aggregates ($sum\{x : P(x) : x\}$) for numeric support, with (inductive) definitions (sets of rules, calculated under the well-founded semantics [8]) and type derivation and checking. This language of IDP is an extension of first-order logic (FO) named FO$(\cdot)^{\text{IDP}}$.

A structure object has the syntactical form

```
structure <name> : <voc name> { <list of interpretations> }
```

The structure must contain a full interpretation of the types of the vocabulary (the domain) and a partial interpretation of the other symbols in the vocabulary. A structure is called two-valued if every symbol is fully interpreted. A model is a structure that satisfies a theory.

A procedure object has the syntactical form

```
procedure <name>(<parameters>) {<instructions>}
```

A procedure has a name, parameters and imperative instructions in Lua. Lua is used as a scripting language in IDP to execute inferences on the above logic objects. Inferences are special procedures built in IDP to reason with these vocabularies, theories and structures (cf. De Cat, Bogaerts, Bruynooghe, and Denecker [4]). In this paper only two of these inferences are used: model expansion and progression [1].

Modelexpansion(T, S) The first inference, model expansion, expands a partial structure I to a model M so that $I \subseteq M$ and $M \models \mathcal{T}$.

For example the vocabulary(V), theory(T) and structure(S) from Listing 1 can be expanded to the three models (M1, M2, M3) from Listing 2:

```
vocabulary V : {
  type Num isa int
  A : Num
  B : Num
}
theory T : V {
  A + B > 8.
}
structure S : V {
  Num = {1..5}
}
procedure main() {
  stdoptions.nbmodels=4;
  printmodels(modelexpand
      (T, S))
}
```

Listing 1: Input

```
structure M1 : V {
    Num = {1..5}
    A = 4
    B = 5
}
structure M2 : V {
    Num = {1..5}
    A = 5
    B = 5
}
structure M3 : V {
    Num = {1..5}
    A = 5
    B = 4
}
```

Listing 2: Output

Progression(T, I_0) The second inference, progression, is an inference that needs a linear time theory and a structure over the same vocabulary. Unlike model expansion, which creates a model with a full planning, progression generates the models of the next time step. We can choose the next state (a 'snapshot') before continuing. This way we can dynamically simulate a linear time theory step by step.

A simulation has two parts. The first inference returns a list of possible initial states with `initialise(T, S)`. The second inference, `progress(T,S`$_i$`)`, returns a list of models.

The models of these two inferences have a special vocabulary: the single state vocabulary. This is a vocabulary where every time parameter in a predicate or function of the original vocabulary is projected away. The model represents the state of the simulation at the current time.

3 IDPD3

IDPD3 is a visualisation library for IDP. It is the successor of ID^P_{Draw} [6]. The goal of this library is to easily visualise structures as a drawing. An example can be found in Figure 2. IDPD3 consists of two parts. The first part creates a description of a drawing from a structure as a JSON string, it is an encoding of the relevant predicates and functions. This transformation is done in the Lua environment of IDP (cf. the left part of Figure 1). The second part is integrated with the IDP Web-IDE [7] and is written in Javascript. It interprets the description to visualise using the d3 visualisation library [2]. The d3 library can visualise any svg primitive that HTML supports. This part corresponds to the right part of Figure 1. Since Lua is always integrated in IDP and Javascript is a widely used language IDPD3 is more platform independent than the previous version ID^P_{Draw} which is written in C++.

To start, we will describe the features implemented in IDPD3. Next we will show how the encoding is done by the library and finally we will decode the input sent from the IDP Web-IDE.

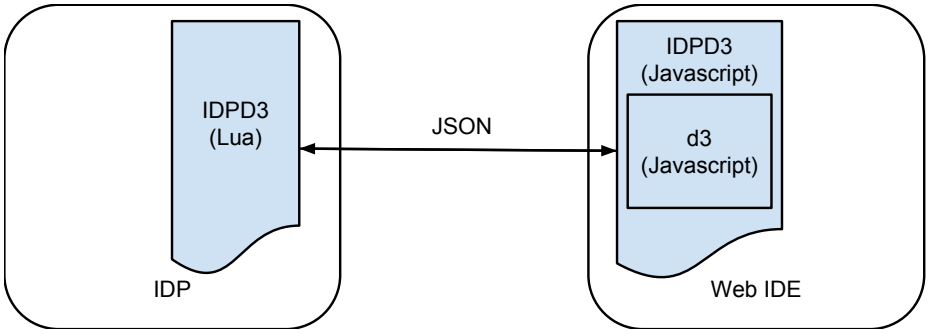

Fig. 1: The software environment that uses IDPD3

3.1 Features

IDPD3 can visualise structures in an interactive way, the current snapshot in a sequence of snapshots that correspond to a model of the linear time theory. This way it can visualise the output of many forms of inferences such as model expansion, minimisation, expanding linear time theories and the simulation of linear time theories. To support this ID^P_{Draw} introduces frames. These are multiple drawings that can be viewed like a slideshow.

In IDPD3 frames are extended with animated transitions that are built in d3. Elements are defined by their keys. When attributes change for an element with the same key between two different frames, they can be animated. This helps the viewer to track the changed elements, even when some frames are

skipped. For example the positions, sizes and colours can be interpolated by d3. Other animations, like morphing a rectangle to a circle, are not implemented, but they could be easily added without modifying the JSON encoding.

Elements can be one of four basic primitives: a rectangle, a circle, text or an image. Each has different attributes. Every primitive has a position (x, y), a z-order and a colour. Rectangles also have a size (width, height). Circles have a radius. Text has a string (label) and a size. Images have a URL, a size(width, height) and no colour.

If the function value is missing for some key the value may default. Colours default to black. The other numeric attributes default to zero, this means that when no size is given, the element will be invisible. When no position is given, the element will be positioned at the edge of the visualisation.

There is also a special primitive to create an undirected visual link between two basic primitives. It has three attributes: link-from, link-to and link-width. The first two declare the keys to the primitives that must be linked. The third declares the width of the link that must be drawn. This enables us to draw graphs just as Kara, *ASPVIZ* and ID^{P}_{Draw}. These tools also have automatic placement functions for graphs. In IDPD3 this is done with a declaration whether a primitive is a node and whether the primitives position is fixed. Declaring as a primitive a node adds it to a force-directed layout implemented in d3.

Finally IDPD3 adds a hook on the elements. When an element is clicked a JSON string is sent to the IDP process. This string contains an identification of the current time-frame and the key of the element that was clicked. This hook is created to support simulated linear time theories.

3.2 Creating drawings

For a visualisation we need a structure for which the vocabulary is an extension of the IDPD3 output vocabulary. The creation of this structure can be done in three ways. The first is to manually specify a structure. The second option is to place the visualisation code directly inside the original theory and expand this theory to a model. The third option is to separate the two logic theories and expand the visualisation theory together with the generated model of the original theory as an extra step. The last approach allows us to separate the concern of the original model from the visualising model. This separation of concerns is one of the core values in software engineering.

In Listing 3 the types of the vocabularies, both for input and for output, are defined. The output predicates and functions are also shown in this listing. These are the symbols that need to be used to create a visualisation.

```
vocabulary V_types {
  type shape constructed from {circ, rect, text, link, img}
  type time isa int
  type key isa string
  type color isa string
  type label isa string
```

```
  type width isa int
  type height isa int
  type order isa int
  type image isa string
}
vocabulary V_out {
  extern vocabulary V_types

  d3_width(time) : width
  d3_height(time) : height
  partial d3_type(time, key) : shape
  partial d3_x(time, key) : width
  partial d3_y(time, key) : height
  partial d3_color(time, key) : color
  partial d3_order(time, key) : order
  partial d3_circ_r(time, key) : width
  partial d3_rect_width(time, key) : width
  partial d3_rect_height(time, key) : height
  partial d3_text_label(time, key) : label
  partial d3_text_size(time, key) : width
  partial d3_img_path(time, key) : image
  partial d3_link_width(time, key) : width
  partial d3_link_from(time, key) : key
  partial d3_link_to(time, key) : key
  d3_node(time, key)
  d3_isFixed(time, key)
}
```

Listing 3: The types vocabulary and the output vocabulary in IDPD3

IDPD3 contains a procedure to transform a structure over the V_out vocabulary to a JSON string. The algorithm does this by looping over every IDPD3 symbol in the output structure and reading every tuple of the predicate or of the graph interpretation of the function. The function mapping is then added to the element with the correct time-frame and key as a key-value pair. Extending this algorithm with a new attribute is easy, it is a new symbol that must be read and the tuples must be transformed to a new key-value pair to add at the correct place.

When this JSON string is sent to the IDP Web-IDE it will be interpreted as an image and it will be visualised. First it is parsed and then interpreted by the online part of the library. The JSON encoding was chosen in such a way that it coincides largely with the data-driven approach of the d3 library.

Application: visualising a structure A structure over an extension of the output vocabulary can be visualised just by transforming it to JSON. For example the structure in Listing 4 has a rectangle with basic attributes a position (2,3), a width(4), a height(5) and with the default colour black. The dots signify places where more elements and attributes can be defined.

```
structure S : idpd3::V_out {
    d3_type = {1, "key", rect(); ...}
    d3_width = {1, "key", 4; ...}
    d3_height = {1, "key", 5; ...}
    d3_x = {1, "key", 2; ...}
    d3_y = {1, "key", 3; ...}
    ...
}
procedure main() {
    visualise(S);
}
```

Listing 4: Example structure

This structure is transformed to the following JSON specification by Lua. This can be done at any time for a two-valued structure over V_out by calling "idpd3:visualise(structure)". This method will transform the structure and print the specification.

```
{"animation": [{"time":1, "elements": [{"key":"key", "type":"rect",
  "y":3, "x":2, "rect_height":"5", "rect_width":"4"}, ...]
, ...}, ...]}
```

The IDP WEB-IDE will filter the specification and visualise it immediately. Thus the IDP WEB-IDE will draw this rectangle (and the other elements).

Application: Transforming a model to a drawing As a general principle in software development different responsibilities should be divided as much as possible. When you have a logic structure and theory that is expanded to a model you might want to visualise it. Without changing the original theory you can create a visualising theory to expand the original model to a model that can be visualised.

For example the structure S is a three by three chessboard grid. The structure is expanded under the theory T to a structure sol and then visualised.

```
vocabulary V {
  type X isa int
  isBlack(X,X)
}
structure S : V {
  X = {1..3}
  isBlack = {(1,2); (2,1); (2,3); (3,2)}
}
vocabulary V_out {
  extern vocabulary V
  extern vocabulary idpd3::V_out
  toKey(X, X) : key
}
```

```
theory T : V_out {
 {
  d3_type(1, toKey(x, y)) = rect.
  d3_x(1, toKey(x, y)) = 4*x - 2.
  d3_y(1, toKey(x, y)) = 4*y - 2.
  d3_rect_width(1, toKey(x, y)) = 4.
  d3_rect_height(1, toKey(x, y)) = 4.
  d3_color(1, toKey(x, y)) = "black" ← isBlack(x, y).
  d3_color(1, toKey(x, y)) = "white" ← ¬isBlack(x, y).
  d3_width(1) = 14.
  d3_height(1) = 14.
 }
}
procedure toKey(x, y) {
    return x.."-"..y;
}
structure S_out : V_out {
    time = {1}
    color = {"black"; "white"}
    width = {1..15}
    height= {1..15}
    X = {1..3}
    toKey = procedure toKey
}
procedure main() {
  local m = merge(S, S_out);
   local sol = onemodel(T, m);
   visualise(sol);
}
```

Application: Comparing theories Another of our basic applications checks whether two theories are (approximately) equivalent on some partial structure. It is designed to highlight the errors of the user (or student) designed theory compared to a given correct theory. The application needs five arguments: a theory the user created (T_{user}), the correct theory(T_{corr}), a visualisation theory(T_{vis}) and two structures that contain as basic information for the user (S) and the basic information for the visualisation (S_{out}).

The application will try to find a model of the first theory that is not satisfied by the second theory. If it finds such a model exists, it is shown as a drawing where the errors are highlighted. Otherwise it will show one of the matching models. To reduce the amount of computing time, a finite number of models can be checked.

This application is used in the examples of the online editor of IDP. The example of Figure 2 can be found at `https://dtai.cs.kuleuven.be/krr/idp-ide/?present=Roster`. There are currently two rules that are wrongly encoded, which is displayed in the images as the rectangles which are coloured

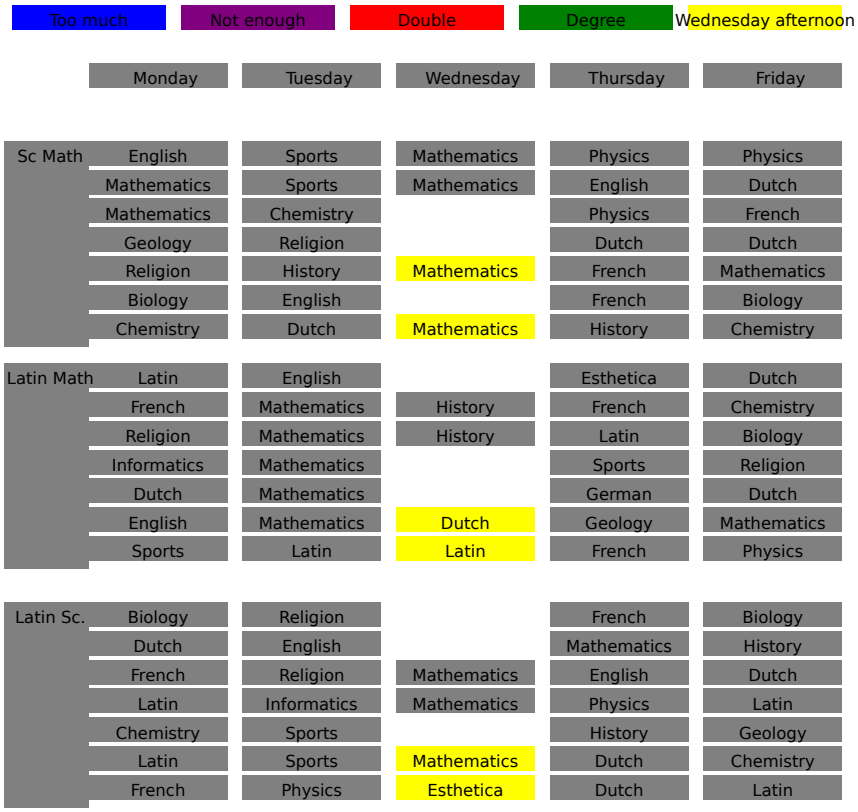

Fig. 2: An example of the images that can be produced with IDPD3

according to the error that is made. The correct rules are supplied in the comments, activating these will remove the coloured rectangles.

3.3 Interpreting interactions

Another extension of ID^{P}_{Draw} in IDPD3 is to support interaction with the user using logic LTC theories and progression inference. After an initial model is created and visualised, IDP will wait for input from the IDP Web-IDE. This input is interpreted by the library and transformed to an input structure. The union of the current snapshot and the input structure is expanded to a structure that holds the current state and the chosen action. This structure is then used to progress to the next state. This way we can create an interactive simulation of an LTC theory. This loop behaves like the Model-View-Controller pattern. Where the output theory creates the View, the input theory behaves as the Controller, and the progression theory handles the Model.

In IDPD3 the input is only click-based: the only predicate that is available in the input vocabulary is `d3_click(Time, Key)`. The input is transformed to an IDP structure by handling every object defined by a time-key pair. For example when an element with key "key" is clicked the drawing program will generate the following specification:

```
[{"time":1,"elements":[{"key":"key","type":"click"}]
```

This is transformed to the following structure:

```
structure S : V {
  time = {1}
  key = {"key"; ...}
  d3_click = {1, "key"}
}
```

Application: Interactive simulation Another application enables the interactive simulation of an LTC theory. This application needs 9 arguments. The first three are the progression theory (T_{prog}), the initial structure (I_{init}) and the output vocabulary (V_{state}). The next three handle the output: the output theory (T_{out}), structure (S_{out}) and vocabulary (V_{out}). The last three handle the input: the theory (T_{in}), structure (I_{in}) and vocabulary (V_{in}).

This application first calculates an initial model M_0 by expanding S_{init} with the theory T_{prog}. This model is merged with the output structure S_{out} and expanded to the visualisation model M_{v0} using T_{out}. This visualisation model is transformed using the library to a JSON string and visualised by the IDP Web-IDE. The application then waits for input.

When input is received it is added to S_{in} creating a more specific structure. This structure is expanded with T_{in} and projected to the actions of the model. The application keeps looping until no new model exists for one of the inferences.

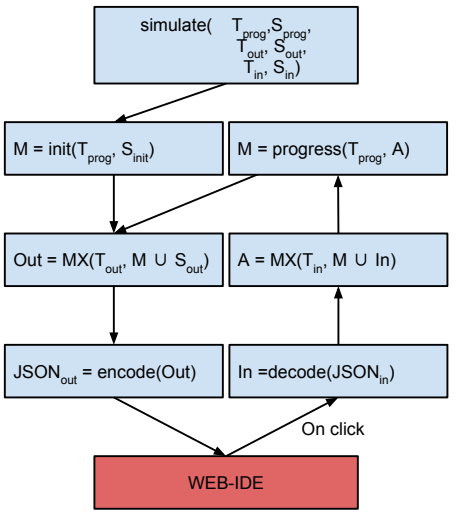

Fig. 3: The MVC loop

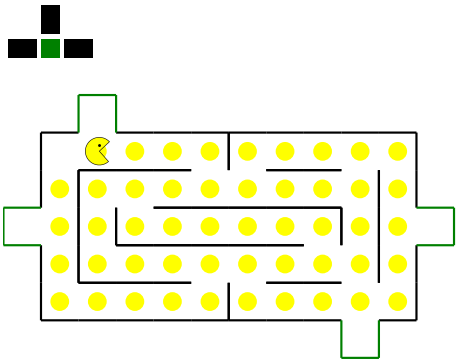

Fig. 4: Pac-Man as a game in IDP

A schematic view of the interactions between the theories and the structures is given in Figure 3.

In this way an interactive game, like Pac-Man (Figure 4), can be made. This application is available at `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=pacman`.

4 Full example

Suppose we have an LTC theory with a single function, count(Time) : Count. There are also three actions: countUp(Time), countDown(Time), setValue(Time, Value). This theory would be declared the following way:

```
LTCvocabulary V_types {
    type Count isa int
    type Time
    Next(Time) : Time
    Start : Time
}
LTCvocabulary V_state { ... }
LTCvocabulary V_action { ... }
LTCvocabulary V { ... }

theory T : V {
 {
  count(Start) = 0.
  count(Next(t)) = v ← setValue(t, v).
  count(Next(t)) = count(t) + 1  ← countUp(t).
```

```
  count(Next(t)) = count(t) - 1  ← countDown(t).
  count(Next(t)) = count(t) ← ¬countUp(t) ∧
      ¬countDown(t) ∧ ∀v : ¬setValue(t, v).
 }
}
```

This theory can be augmented with IDPD3 by adding the visualisation theories. The output vocabulary is the union of the single-state vocabulary[1] and the IDPD3 output-vocabulary. Helper functions and predicates can be added in this vocabulary. The output theory declares two text elements: the current count and a label to count up. The input vocabulary is the union of the single-state vocabulary and the IDPD3 input-vocabulary. The input theory declares two rules to convert the clicks from the user: when the "Count up" text is clicked the counter is incremented, when the count itself is clicked the counter resets to zero.

```
LTCvocabulary V_d3 {
    extern vocabulary V_types
    extern vocabulary idpd3::V_types
    toLabel(Count) : label
    countLabel : label
}
vocabulary V_d3_out { ... }
vocabulary V_d3_in { ... }

theory T_out : V_d3_out {
 {
  d3_width(1) = 10.
  d3_height(1) = 10.

  d3_type(1, "label") = text.
  d3_x(1, "label") = 1.
  d3_y(1, "label") = 1.
  d3_text_size(1, "label") = 1.
  d3_text_label(1, "label") = toLabel(count).

  d3_type(1, "button") = text.
  d3_x(1, "button") = 1.
  d3_y(1, "button") = 5.
  d3_text_size(1, "button") = 1.
  d3_text_label(1, "button") = "Count up".
}}
theory T_in : V_d3_in {
 {
  countUp ← d3_click(1, "button").
  setValue(0) ← d3_click(1, "label").
}}
```

[1] V_ss is an automatically generated vocabulary where Time is projected away from the vocabulary.

Finally the basic structure and the Lua-scripting environment are declared. An IDP procedure is used to automatically convert from a number to a string. And due to quirks in IDP with types there is one structure that is projected to the different structures needed for the application. It is possible to disambiguate it, at the cost of using more advanced features in IDP. The full source code is available for testing at `https://dtai.cs.kuleuven.be/krr/idp-ide/?present=Count`.

```
structure S : V {
    Count = {0..100}
    Start = 0
}
structure S_d3 : V_d3_ss {
    Count = {0..100}
    time = {1}
    key = {"label"; "button"}
    //label is autogenerated by procedure output
    width = {0..20}
    height = {0..20}
    countLabel = "Count up"
    toLabel = procedure toText
}
procedure main() {
    stdoptions.splitdefs = false;
    stdoptions.postprocessdefs = false;
    stdoptions.cpsupport = false;
    stdoptions.xsb = false;
    idpd3_browser:setLogLevel(0);
    local go = idpd3_browser:createLTC(
        T, S, V_state_ss,
        T_in, S_d3, V_d3_in,
        T_out, S_d3, V_d3_out);
    local lastState = go();
}
```

5 Related work

Multiple systems have already been created to visualise logic programs. The main systems are *ASPVIZ* [3], ID^{P}_{Draw} [6] and Kara [5].

ASPVIZ is one of the first visualisation tools for logic programs. This program joins the original logic program with a logic visualisation program before expanding it with an ASP solver. As it is one of the early visualisation tools created, it only supports basic primitives and visualising multiple frames. *ASPVIZ* supports saving images as svg, like IDPD3 and Kara.

ID^{P}_{Draw} is another visualisation tool. It is the predecessor of IDPD3. The main improvement over ID^{P}_{Draw} is that the program is built on the core of IDP itself. We use an IDP vocabulary, which helps during the construction of

visualisation theories with grammar checking and debugging. Additionally, the transformation of the structures are written in Lua, the imperative language that drives the inferences. This means that while ID^P_{Draw} is capable of providing interactivity, it is limited to direct keyboard interaction with IDP and spawning multiple windows. IDPD3 has extended this to clicking with the mouse on drawn items.

One of the newest visualisation tool is Kara. This system is part of the SeaLion IDE. It supports an almost full superset of both $ASPVIZ$ and ID^P_{Draw}, lacking only the ability to show short non-interactive frames. Like ID^P_{Draw} and IDPD3, Kara supports ordering elements by a z-axis. One of the strengths of Kara is that it supports some higher-level specifications that are not supported by the other main systems. It can generate layouts for graphs and visualise an arbitrary answer set model as a hyper-graph.

6 Future work

In the future we would like to extend the framework of interactive animations to more practical applications. For this some basic functionality must be added: keyboard interaction and text fields are a standard way of entering information.

Another change we would like to implement is generalising the vocabulary of both input and output. Currently it takes a minimal but non-zero time to implement a new attribute. If this is done new attributes can be added without changing the library. This would decrease the time invested in maintaining the library in the same way as argued for the d3 visualisation library [2].

However to really support practical applications we should need to move away from visualisations and support forms. A new library that uses many of the ideas currently implemented in IDPD3, would be created for this.

7 Conclusion

In this paper we presented IDPD3 as the successor of ID^P_{Draw}. The main feature of IDPD3 is the possibility to visualise an interactive simulation of a linear time theory. Smaller features include the animation of basic primitives and the tutorial application. Due to it's integration with both IDP as a library and the IDP Web-IDE developing visualisations for IDP is easier and platform independent. The maintenance of the library should be easier than before. In the future we would like to implement the library for more practical applications involving data entry and perhaps forms.

References

[1] Bart Bogaerts, Joachim Jansen, Maurice Bruynooghe, Broes De Cat, Joost Vennekens, and Marc Denecker. "Simulating Dynamic Systems Using Linear Time Calculus Theories". In: *TPLP* 14 (4-5 July 2014), pp. 477–492. ISSN: 1475-3081. URL: http://journals.cambridge.org/article_S1471068414000155.

[2] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D3: Data-Driven Documents". In: *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). URL: `http://vis.stanford.edu/papers/d3`.

[3] Owen Cliffe, Marina De Vos, Martin Brain, and Julian Padget. "ASPVIZ: Declarative visualisation and animation using answer set programming". In: *Logic Programming*. Springer, 2008, pp. 724–728.

[4] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. "Predicate Logic as a Modelling Language: The IDP System". In: *CoRR* abs/1401.6312 (2014).

[5] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. "Kara: A system for visualising and visual editing of interpretations for answer-set programs". In: *Applications of Declarative Programming and Knowledge Management*. Springer, 2013, pp. 325–344.

[6] IDP*Draw: Finite structure visualization*. `http://dtai.cs.kuleuven.be/krr/software/visualisation`. 2012.

[7] *The IDP web-IDE*. `https://dtai.cs.kuleuven.be/krr/idp-ide/`. 2014.

[8] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. "Unfounded Sets and Well-Founded Semantics for General Logic Programs". In: *PODS*. ACM, 1988, pp. 221–230. ISBN: 0-89791-263-2.